REMARKS

1. The present invention, as claimed

It is conventional that when a process encounters a breakpoint, a debugger temporarily removes the breakpoint, an instruction for which the breakpoint was originally substituted is executed, and then the breakpoint is immediately replaced so that if the instruction is again encountered the breakpoint will fire. The present invention deals with a problem that arises when two threads encounter the same breakpoint and the debugger temporarily removes the breakpoint for one of the threads while processing of the breakpoint for the second thread is still pending. That is, when the processing of the breakpoint occurs for the second thread, it is problematic if the breakpoint has been temporarily removed because of the first thread. Page 5, lines 18 through 20. In such a case, the absent breakpoint is referred to in the present application as a "zombie" breakpoint and the problem is referred to as the "zombie breakpoint problem."

Claim 1 states that a breakpoint data structure is checked to determine if the data structure has an entry for a breakpoint known to a debugging process for a certain address where a breakpoint fired. Then, the claim goes on, there is a second step of verifying if a breakpoint condition continues to exist at the address where the breakpoint fired. As the claim states, this second step occurs if no entry is found by the first step. If the breakpoint does not exist, i.e., because it has been temporarily removed, the claim states that the breakpoint is identified as a zombie breakpoint. Independent claims 7 and 13 have similar language. This is in contrast to the conventional result, in which it is incorrectly concluded, due to the absence of the removed breakpoint, that the exception was not caused by a breakpoint.

2. Rejections under 35 USC 112, second paragraph

Claims 1-18 stand rejected under 35 USC 112, second paragraph. The Office action asserts that the independent claims 1, 7 and 13 are indefinite because i) there is no description of what the checking function was or did, ii) there is no description of how the determining is conducted, and iii) the determining is only an intended action. Applicant respectfully contends the rejections are improper for the following reasons.

Notwithstanding the following remarks, Applicant understands there are profound time pressures attending the examination process. Understandably, this sometimes leads to abbreviated statements of rejection. If the rejections have some substance that Applicant is overlooking, Applicant will gladly respond more fully to the rejections upon clarification and citation of more specific authorities. Further, Applicant herein includes a completed form requesting a telephone interview to discuss the rejections. It is Applicant's hope that a discussion of the issues may be helpful. In this same regard, Applicant encourages Examiner to suggest claim language to improve the clarity or precision of the language used. MPEP 2173.02.

Regarding reasons Applicant contends the rejections are improper, Applicant is unaware of any requirement, and the Office action cites no authority for a requirement, that a claim must describe what a claimed function is or does. Even if there were a requirement for a claim to describe what a claimed function does or is, claims 1, 7 and 13 in the present case do describe what the checking is and does. The checking is to determine if the data structure has an entry for a breakpoint known to a debugging process for a certain address where a breakpoint fired.

Second, Applicant is unaware of any requirement, and the Office action cites no authority for a requirement, that a claim may not set out an intended action. Even if there were a prohibition against a claim setting out an intended action, claims 1, 7 and 13 set out more than intended actions. That is, the second step in claim 1, for example, states what is done if no entry is found by the checking of the data structure, which clearly follows as a result of the determination referred to in the first step. Claims 7 and 13 have similar language.

Third, Applicant is unaware of any requirement, and the Office action cites no authority for a requirement, that a claim must describe how a step is conducted. Applicant contends that there is no requirement that a claim must state how a function is done, or what is a function, at least partly because this could lead to an endless series of rejections based on further demands for how or what. That is, if Applicant claimed "touching green widgets to see if they are made of metal," an endless series of how or what questions could always be posed. First there could be, "How is the touching done?" And if the Applicant amended the claim to state, "wherein the touching is by a magnet," the question could then be posed "How is the touching by a magnet done?" This could continue ad infinitum. Similarly, if the question were posed, "What is it to touch a green widget with a magnet, or what does that touching do?" Applicant might amend the claim to state, "wherein the touching by a magnet is to cause an attraction of a certain force between the magnet and the widget if the widget is a certain metal." But the question could again be posed, "What else does it do when you touch a green widget by a magnet to cause an attraction of a certain force between the magnet and the widget if the widget is a certain metal?" The real issue should be whether Applicant has clearly stated something that is different and nonobvious with respect to the prior art, not whether Applicant has stated a certain amount of specific detail in the claims about how or what. Breadth of a claim is not to be equated with indefiniteness. MPEP 2173.04 (citing In re Miller, 441 F.2d 689, 169 USPQ 597 (CCPA 1971)).

3. Rejections under 35 USC 102(e) based on Bhattacarya

Claims 1-18 stand rejected under 35 USC 102(e) as being anticipated by U.S. patent 6,708,326 ("Bhattacarya"). Applicant respectfully contends the claims are patentably distinct, for the following reasons. To provide a context for the Bhattacarya passage cited in the Office action, consider the following teachings of Bhattacarya.

Bhattacarya primarily concerns improvement in speed of handling breakpoints. See Bhattacarya, col. 2, lines 18-29; col. 4, lines 22-33 (describing that breakpoint processing may be faster by avoiding the single-stepping conventionally done in the processing of breakpoints). Bhattacarya does mention a multi-thread problem of missed breakpoints. Bhattacarya, col. 2, lines 30-34. Bhattacarya teaches that this is dealt with by stopping all other threads when a first thread encounters a breakpoint, so that certain settings may be changed. Bhattacarya, col. 4, lines 12-21. But claim 1 in the present case states that a data structure is checked for an entry for a breakpoint known to a debugging process, and, if there is no entry, the method includes verifying if a breakpoint condition continues to exist at the address where the breakpoint fired. Claims 7 and 13 have similar language. Bhattacarya does not teach this.

Bhattacarya's method of handling breakpoints involves the following steps, which have conventionally been used in connection with putting breakpoint instructions into a debugee program instruction stream:

> (i) storing at respective addresses of said memory a sequence of processor instructions to be processed by the processor;
> (ii) replacing one of said processor instructions in said sequence with a break instruction, wherein in step (i) the one of said processor instructions was stored in a certain one of the addresses, and wherein the replacing includes storing the break instruction in the memory at the certain address in place of said one processor instruction;

> (iii) supplying said sequence of processor instructions, including the break instruction in place of the said one processor instruction, to said processor . . . Bhattacarya, Claim 1.

According to Bhattacarya, these steps are combined with breakpoint register means to provide an advantageous new arrangement, in which single-stepping is eliminated after encountering a breakpoint. That is, the above steps (i) through (iii) cause a breakpoint to occur as a result of a break point instruction inserted in the debugee instruction stream. After encountering a breakpoint instruction in a debugee program stream, and after reinserting the debugee program instruction in place of the break point instruction, it is conventional to execute just that one debugee program instruction in a single-step fashion and then *immediately* reinsert the breakpoint instruction. See Bhattacarya, col. 3, lines 25-44 (describing conventional practice). It is conventional to immediately reinsert, i.e., "restore," the breakpoint instruction, in case the debugee program would otherwise again encounter the same debugee program instruction. However, according to the Bhattacarya invention:

> The restoration of the breakpoint instruction after execution past a breakpoint (after putting back the original instruction) is not done immediately. This enables the omission of the single-step after restoring the original instruction in the sequence described earlier. Instead, at that time, a breakpoint register is used to set an instruction breakpoint at that address, and then a flag (e.g. the RF flag in Intel) is set in the processor to ensure that the original instruction can execute without faulting right away, and yet cause a fault the next time the same point is hit. The next time the debugger gets control (say, when another breakpoint is hit) in the same process context, the breakpoint instruction can be put back (herein this is called "hardening" of the breakpoint), so that the breakpoint register is free for the next use. Bhattacarya, col. 3, lines 11-24.

See also, Bhattacarya, claim 1, step (iv).

It should be understood that the use of a breakpoint register to cause a break point in a debugee program has conventionally been an *alternative* to putting breakpoint instructions into the debugee program instruction stream. However, according to Bhattacarya, the breakpoint register means is used for a purpose different than the prior art. Bhattacarya, col. 3, lines 9-10. That is, Bhattacarya teaches using a breakpoint register and putting breakpoint instructions into the debugee program instruction stream *in cooperation with one another*.

The Office action relies upon Bhattacarya, col. 6, line 45 - col. 7, line 35, for the rejection of claims 1, 7 and 13. In this regard, the Office action contends that Bhattacarya, col. 6, line 45-67 teaches the first two steps of claim 1, for example. The Office action specifically quotes the language of the first of Bhattacarya's eight steps at col. 6, lines 61-63.

In the cited passage, Bhattacarya describes eight steps of his method and then describes an example application of the steps for a first and second thread running a program. In Bhattacarya's example, a) the first thread hits a line in the program code in which a first breakpoint has been inserted, b) the breakpoint is removed from the line and the line's address is loaded in the breakpoint register, then, c) just as the first thread is about to execute the first breakpoint and continue beyond, a context switch occurs, i.e., the second thread resumes, and d) the second thread then hits a second, "different" breakpoint, i.e., in a different line of the program. Bhattacarya, col. 7, lines 27 - 30,

This example application puts into context the first of Bhattacarya's eight steps, which the Office action quotes. That is, when the second thread hits the second breakpoint, the breakpoint register is in use for the first breakpoint. Bhattacarya, col. 6, line 45 - col. 7, line 35 (step 1). But the breakpoint register must be used for the second breakpoint. To free up the breakpoint register, the first breakpoint is reinserted into the program. Id. (step 2, referred as "hardening the last breakpoint"). If the breakpoint register had not been in use, step 2 could have been skipped. Id. (step 1). Then the second breakpoint is recorded as the last active breakpoint. Id. (step 3). The address of the line of program code for the second breakpoint is loaded into the breakpoint register. Id. (step 4). Then the instruction is replaced in the program instructions, i.e., at the location where the second breakpoint was just encountered. Id. (step 5). Then the instruction pointer for the second thread is set to point at the replaced instruction. Id. (step 6). The breakpoint exception is temporarily suppressed, to ensure that the instruction can execute without faulting right away, and yet cause a fault the next time the same point is hit. Id. (step 7). Then the program is permitted to continue for the second thread. Id. (step 8).

With the above explanation, it should be understood that Bhattacarya does not teach what is claimed in the present case. That is, Bhattacarya's cited teaching about eight steps of Bhattacarya's invention, about context switching, and about an operating system saving/restoring debug register context across process context switches does not teach checking a data structure for an entry for a breakpoint known to a debugging process, and, if there is no entry, verifying if a breakpoint condition continues to exist at the address where the breakpoint fired, as stated in claim 1 in the present case. (Claims 7 and 13 have similar language.)

The question in step 1 of Bhattacarya, "If the breakpoint register is in use, is there an earlier breakpoint to harden . . .?" decidedly does *not* teach checking a data structure for an entry for a breakpoint known to a debugging process, and, if there is no entry, verifying if a breakpoint condition continues to exist at the address where the breakpoint fired, as stated in claim 1 in the present case. (Claims 7 and 13 have similar language.) In the first place, step 1 in Bhattacarya is not performed on the condition that there is no data structure entry for a breakpoint known to a debugging process, as is the verifying in claims 1, 7 and 13. And Bhattacarya's teaching about ". . . is there is an earlier breakpoint to harden . . .?" is not about whether a breakpoint condition continues to exist at the address where the breakpoint fired, as stated in claims 1, 7 and 13 in the present case. As explained above, Bhattacarya's teaching is about whether the breakpoint register needs to be freed up to save the address of a new breakpoint that has just been encountered.

Also, it should be understood, from the above explanation, that the cited statement of Bhattacarya does not teach identifying the breakpoint as a zombie breakpoint if the breakpoint condition does not exist, as stated in claims 1, 7 and 13 in the present case. In the cited statement that the Office action contends teaches the last step of claim 1, Bhattacarya describes a context switch back to the first thread at the original point where the first thread was about to execute the first breakpoint, which is at a different place in the program than the second breakpoint. Bhattacarya, col. 7, lines 30-32 ("In this situation, context switches to the original point, executing the breakpoint instruction and thus enters the breakpoint handler again."). Thus, Bhattacarya teaches at col. 7, lines 33-35, ". . . since the debug register settings will still [be] present, it is possible to distinguish this situation from a genuine breakpoint and simply ignore it." That is, in this statement Bhattacarya is asserting that when the context switches back to the first thread and the breakpoint handler is thereby entered again, the breakpoint handler does not confuse the first breakpoint, which was encountered by the first thread, with the second breakpoint, which was encountered by the second thread. Bhattacarya clarifies this assertion in the following statement, "To do this, the other threads might need to be stopped and resumed so that the corresponding processor registers get refreshed with the changes to the debug register context settings for the process. It is [expected] that the operating system saves/restores debug register context across process context switches. If this happens at a thread level, then the debug register context changes for setting the new instruction breakpoint will have to be effected on all the thread contexts for that process." Bhattacarya, col. 7, lines 18-26. None of this teaches identifying a breakpoint as a zombie breakpoint if the breakpoint condition does not exist, as stated in the last step of claim 1 in the present case, or as similarly stated in claims 7 and 13.

4. Rejections under 35 USC 102(e) based on Bates

Claims 1-18 stand rejected under 35 USC 102(e) as being anticipated by U.S. patent 6,658,650 ("Bates"). Applicant respectfully contends the claims in the present case are patentably distinct. Most notably, claims 1, 7 and 13 in the present case concern detecting a breakpoint that is absent, i.e., a "zombie breakpoint," whereas Bates concerns detecting when to ignore breakpoints that exist but do not apply.

Bates refers to a global breakpoint as a type of control point. Bates, col. 2, line 30. The passage cited in the Office action states ". . . at block 90 . . . it is determined whether a hit control point is a service entry point." Bates, col. 8, lines 35-37. The distinction Bates is making here is that some control points may exist that are not what he calls service entry points. See Bates, col. 7, lines 17- 23 ("First, with respect to setting a control point, the breakpoint manager starts in block 80 by determining whether the control point to be created is a service entry point. If not, the control point is set in a conventional manner in block 82. If, however, the request is to set a service entry point, control passes to block 84 to retrieve the opcode in the program under debug to be replaced.").

To understand the rest of the cited passage, the context is helpful. Bates describes the need addressed by the Bates invention as follows. "[A] global breakpoint in a computer program causes any active job that uses the computer program to stop when the global breakpoint is encountered . . . Under ideal circumstances, this mechanism can permit a user to gain control of a job to be debugged . . . However, under normal operating conditions, global breakpoints can significantly degrade the performance and availability of a computer since all jobs that hit the global breakpoint, even those not under debug, will trigger the breakpoint. Furthermore . . . a breakpoint in the computer program may be hit by all jobs . . . regardless of whether the users that own the jobs are actively debugging the computer program . . . execution of the job that hit the breakpoint is suspended, and the encounter is reported to the user performing debugging. If the owner of the job is not performing the debugging, however, the job essentially appears hung or stalled to the owner until it is restarted by the user performing the debugging. As such, global breakpoints can be significantly disruptive in many computer environments." Bates, col. 2, lines 33-57.

The passage cited in the Office action continues, "if the hit control point is a service entry point." Bates, col. 8, line 39. It appears that Bates is making the point here that a hit control point might not be a breakpoint, in which case it is not the type of control point that Bates is concerned about. The cited passage continues, "control passes to block 94 to determine whether the current job is under debug." Bates, col. 8, lines 40-41. Presumably, this is because if a job is already under debug the debug service function does not need to be called again.

The cited passage continues, " If not, control passes to block 96 to determine whether the user ID that owns the current job is the same as that stored in the control point table for the service entry point. If so, the service entry function component is called in block 98." Bates, col. 8, lines 41-45. This provides a solution to the Bates problem description, " . . . all jobs that hit the global breakpoint, even those not under debug, will trigger the breakpoint. Furthermore . . . a breakpoint in the computer program may be hit by all jobs . . . regardless of whether the users that own the jobs are actively debugging the computer program." Bates, col. 2, lines 42-49. That is, if a job triggers a breakpoint but the job is not owned by a user who is debugging the job, Bates teaches that in this case the breakpoint should not call the debug service entry function. Thus, Bates is teaching when to ignore a breakpoint *that exists* but should not apply. This is different than, and does not teach, the claim 1 in the present case, which identifies an *absent* breakpoint (a zombie breakpoint), i.e., for which no entry is found in a data structure, by verifying no breakpoint condition continues to exist at the address where the breakpoint fired. Claims 7 and 13 have similar language.

PRIOR ART OF RECORD

Applicant has reviewed the prior art of record cited by but not relied upon by Examiner, and assert that the invention is patentably distinct.

REQUESTED ACTION

Applicant contends that the invention as claimed in accordance with amendments previously submitted is patentably distinct, and hereby requests that Examiner grant allowance and prompt passage of the application to issuance.

Respectfully submitted,

Anthony VS England

Anthony V. S. England
Attorney for Applicants
Registration No. 35,129
512-477-7165
a@aengland.com

cc: Applicant Initiated Interview Request